Mixed-Precision Orthogonalization Scheme and Adaptive Step Size for Improving the Stability and Performance of CA-GMRES on GPUs

Ichitaro Yamazaki(✉), Stanimire Tomov, Tingxing Dong, and Jack Dongarra

Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, USA
iyamazak@icl.utk.edu

Abstract. The Generalized Minimum Residual (GMRES) method is a popular Krylov subspace projection method for solving a nonsymmetric linear system of equations. On modern computers, communication is becoming increasingly expensive compared to arithmetic operations, and a communication-avoiding variant (CA-GMRES) may improve the performance of GMRES. To further enhance the performance of CA-GMRES, in this paper, we propose two techniques, focusing on the two main computational kernels of CA-GMRES, tall-skinny QR (TSQR) and matrix powers kernel (MPK). First, to improve the numerical stability of TSQR based on the Cholesky QR (CholQR) factorization, we use higher-precision arithmetic at carefully-selected steps of the factorization. In particular, our mixed-precision CholQR takes the input matrix in the standard 64-bit double precision, but accumulates some of its intermediate results in a software-emulated double-double precision. Compared with the standard CholQR, this mixed-precision CholQR requires about 8.5× more computation but a much smaller increase in communication. Since the computation is becoming less expensive compared to the communication on a newer computer, the relative overhead of the mixed-precision CholQR is decreasing. Our case studies on a GPU demonstrate that using higher-precision arithmetic for this small but critical segment of the algorithm can improve not only the overall numerical stability of CA-GMRES but also, in some cases, its performance. We then study an adaptive scheme to dynamically adjust the step size of MPK based on the static inputs and the performance measurements gathered during the first restart loop of CA-GMRES. Since the optimal step size of MPK is often much smaller than that of the orthogonalization kernel, the overall performance of CA-GMRES can be improved using different step sizes for these two kernels. Our performance results on multiple GPUs show that our adaptive scheme can choose a near optimal step size for MPK, reducing the total solution time of CA-GMRES.

1 Introduction

The cost of executing software can be modeled by a function of its computational and communication costs (e.g., in terms of required cycle time or energy

M. Daydé et al. (Eds.): VECPAR 2014, LNCS 8969, pp. 17–30, 2015.
DOI: 10.1007/978-3-319-17353-5_2

consumption). For instance, the computational cost can be modeled based on the number of required floating point operations (flops), while the communication includes the synchronization and data transfer between the parallel processing units, as well as the data movement through the levels of the local memory hierarchy. On modern computers, communication is becoming increasingly expensive compared to computation. It is critical to take this hardware trend into consideration when designing high-performance software for new and emerging computers.

The Generalized Minimum Residual (GMRES) method [6] is a popular Krylov subspace projection method for solving a large-scale nonsymmetric linear system of equations. To address the current hardware trend, we studied a communication-avoiding variant of GMRES [5] on multicore CPUs with multiple GPUs [8]. Our experimental results demonstrated that CA-GMRES can obtain the speedups of up to two by avoiding some of the communication on such shared-memory computer architectures. Our experimental results also showed that both the performance and numerical stability of CA-GMRES depends on the two computational kernels, the orthogonalization (*Orth*) and matrix powers kernels (*MPK*). For example, compared with other orthogonalization schemes, the Cholesky QR (CholQR) factorization [7] obtained a superior performance based on the optimized BLAS-3 GPU kernels. Unfortunately, when the input matrix is ill-conditioned, CholQR can be numerically unstable, and CA-GMRES may not converge even with reorthogonalization. We also found that depending on the sparsity pattern of the coefficient matrix, *MPK* can be slower than the standard sparse-matrix vector multiply (*SpMV*) due to the computational and/or communication overheads traded for reducing the communication latency. This is especially true in CA-GMRES, where a relatively large step size is preferred by *Orth*.

To address the aforementioned limitations of CA-GMRES, in this paper, we first design and study a mixed-precision variant of CholQR that takes the input matrix in the standard 64-bit double precision but accumulates some of its intermediate results in software-emulated double-double precision [4]. Compared with the standard CholQR, our mixed-precision CholQR increases the computational cost by 8.5× but the increase in its communication cost is less significant. Since the computation is becoming less expensive compared to the communication on new and emerging computers, we hope to improve the overall numerical stability of CA-GMRES using the higher-precision without a significant increase in the orthogonalization time. Case studies on different GPUs demonstrate that this mixed-precision CholQR can improve not only the overall stability of CA-GMRES but also, in some cases, its performance by allowing a larger step size, avoiding the reorthogonalization, and improving the solution convergence rate. We then study an adaptive scheme that uses different step sizes for *MPK* and *Ortho* and dynamically adjusts the step size of *MPK* at run time. We demonstrate that our adaptive scheme can find a near optimal step size based on the static input parameters and the performance measurements gathered during the first restart loop, and reduce the total solution time of CA-GMRES.

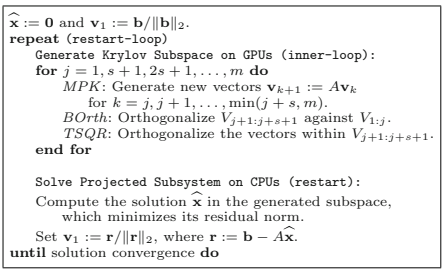


Fig. 1. CA-GMRES(s,m) pseudocode.

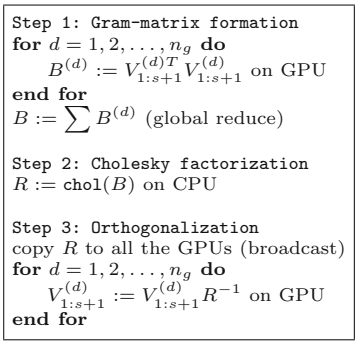


Fig. 2. CholQR pseudocode.

The rest of the paper is organized as follows: In Sect. 2, we first review the CA-GMRES, *MPK*, and CholQR algorithms, and present their implementations on the multicore CPUs with multiple GPUs. Then in Sect. 3, we describe the mixed-precision CholQR and its implementation with the GPU. The performance of the mixed-precision CholQR and its effects on the performance of CA-GMRES are also presented in this section. Next, in Sect. 4, we describe our adaptive scheme for the *MPK*'s step size and present its effectiveness in selecting the near-optimal step size. We provide final remarks in Sect. 5.

2 Communication-Avoiding GMRES

The Generalized Minimum Residual (GMRES) method [6] is a popular Krylov subspace projection method for solving a nonsymmetric linear system of equations, $Ax = b$. The GMRES's j-th iteration generates the $(j+1)$-th Krylov basis vector $\mathbf{v}_{j+1}$. This is done through a sparse matrix-vector multiply (*SpMV*) with the previously-generated basis vector $\mathbf{v}_j$, followed by the orthonormalization (*Orth*) of the resulting vector against all the previously-generated basis vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_j$. As the iteration proceeds, this explicit orthogonalization of the basis vectors becomes increasingly expensive in terms of both computational and storage requirements.

To avoid the expensive costs of generating a large projection subspace, GMRES iteration is restarted after computing a fixed number $m+1$ of basis vectors. Before restart, GMRES updates the approximate solution $\widehat{\mathbf{x}}$ by solving a least-squares problem $\mathbf{g} := \arg\min_{\mathbf{t}} \|\mathbf{c} - H\mathbf{t}\|$, where $\mathbf{c} := V_{1:m+1}^T(\mathbf{b} - A\widehat{\mathbf{x}})$, $H := V_{1:m+1}^T A V_{1:m}$, $\widehat{\mathbf{x}} := \widehat{\mathbf{x}} + V_{1:m}\mathbf{g}$, and $V_{1:m}$ is the matrix consists of the column vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m$. Compared with the coefficient matrix A, the projected matrix H, a by-product of the orthogonalization procedure, is smaller in dimension (i.e., $m \ll n$) and is in a Hessenberg form. Hence, the least-squares problem can be efficiently solved, requiring only about $3(m+1)^2$ flops. On the other hand, for an n-by-n matrix A with $nnz(A)$ nonzeros, *SpMV* and *Orth* require a total of about $2m \cdot nnz(A)$ and $2m^3 n$ flops over the m iterations, respectively

(i.e., $n, nnz(A) \gg m$). Hence, the solution time of GMRES is often dominated by the first step of generating the basis vectors. To accelerate the solution process using GPUs, we distribute the coefficient matrix A and the basis vectors $V_{1:m+1}$ in a 1D block row format among the GPUs. We then generate these basis vectors on the GPUs, while the least-square problem is solved on the CPUs.

Even with a single GPU, both *SpMV* and *Orth* require communication to move the data through the memory hierarchy of the GPU, while with multiple GPUs, additional communication is needed among the GPUs. CA-GMRES [5] aims to reduce this communication by redesigning the algorithm to replace *SpMV* and *Orth* with three new kernels – matrix powers kernel (*MPK*), block orthogonalization (*BOrth*), and tall-skinny QR (*TSQR*) – that generate and orthogonalize a set of s basis vectors at once. By avoiding the communication, even with one GPU, CA-GMRES can obtain a speedup of up to two [8]. Figure 1 shows the pseudocode of CA-GMRES (s, m). A more detailed description of our implementation can be found in [8].

In the rest of this section, we review the two main computational kernels of CA-GMRES, *TSQR* and *MPK*, improving whose performance is the focus of this paper.

2.1 Cholesky QR Factorization

To orthonormalize the tall-skinny matrix $V_{1:s+1}$ with $s+1$ columns, we focus on *TSQR* based on the Cholesky QR (CholQR) factorization [7]. To describe our implementation of CholQR on multicore CPUs with multiple GPUs, we use $V_{1:s+1}^{(d)}$ to denote the local submatrix of $V_{1:s+1}$, distributed to the d-th GPU, and n_g is the number of available GPUs. To orthogonalize the $s+1$ vectors $V_{1:s+1}$, CholQR first forms the Gram matrix, $B := V_{1:s+1}^T V_{1:s+1}$, through the local matrix-matrix product $B^{(d)} := V_{1:s+1}^{(d)T} V_{1:s+1}^{(d)}$ on the GPU, followed by the reduction $B := \sum_{d=1}^{n_g} B^{(d)}$ on the CPU. Next, the Cholesky factor R of B is computed on the CPU. Finally, the GPU orthogonalizes $V_{1:s+1}$ by a triangular solve $V_{1:s+1}^{(d)} := V_{1:s+1}^{(d)} R^{-1}$. Hence, all the required GPU-GPU communication is aggregated into a pair of messages between the CPU and GPUs, while all the GPU computation is based on BLAS-3. As a result, both intra and inter GPU communication can be optimized. Figure 2 shows these three steps of CholQR. Unfortunately, the condition number $\kappa(B)$ of the Gram matrix B is the square of the condition number $\kappa(V_{1:s+1})$ of the input matrix $V_{1:s+1}$ (i.e., $\kappa(B) = \kappa(V_{1:s+1})^2$). This often causes numerical instability, especially in CA-GMRES, where even using the Newton basis [1], the vector $\mathbf{v}_j$ can converge to the principal eigenvector of A, and $\kappa(V_{1:s+1})$ can be large.

2.2 Matrix Powers Kernel

For *SpMV* on multiple GPUs, the communication of the distributed vector elements through the PCI Express bus could become a bottleneck. To reduce this bottleneck, given a starting vector $\mathbf{v}_j$, *MPK* first communicates all the

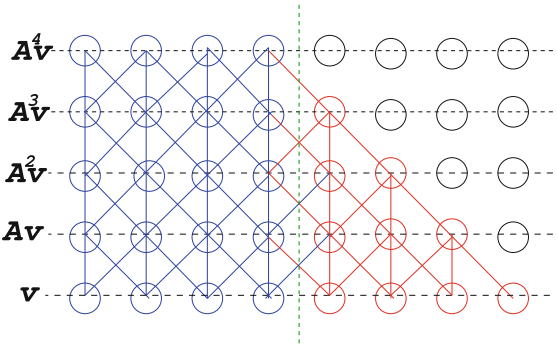


Fig. 3. Illustration of *MPK* for a tridiagonal matrix A with the starting vector $\mathbf{v}$ and the step size $s = 4$. The blue circles represent the local elements of the vectors to be computed on this GPU and the red circles are the required non-local vector elements. The GPU first communicates the red elements of $\mathbf{v}$ on the s-level overlap, then independently performs *SpMV* (Color figure online).

required vector elements of $\mathbf{v}_j$ on the s-level overlap at once so that each GPU can independently compute the local components of the s matrix-vector products $A\mathbf{v}_j, A^2\mathbf{v}_j, \ldots, A^s\mathbf{v}_j$ without further communication [2]. Figure 3 illustrates our implementation of *MPK* for a tridiagonal matrix A. As a result, *MPK* reduces the communication latency by a factor of s, but introduces the overheads to store, communicate, and perform computation on the s-level overlap. Though *MPK* often improve the performance of *SpMV* using a small step size s, its optimal step size may be much smaller than that of *BOrth* or *TSQR* due to the overheads associated with *MPK*. See [8] for the detailed discussion of our implementation of *MPK* and its performance.

3 Mixed-Precision CholQR

To improve the numerical stability of CholQR in the working 64-bit double precision, we use a software-emulated quadruple precision at the first two steps of CholQR, while the last step is in the working precision. This is motivated by the fact that the condition number of the Gram matrix B is the square of the condition number of the input matrix $V_{1:s+1}$. Hence, the numerical stability should be improved by using the higher-precision arithmetic for forming and factorizing the Gram matrix (see [9] for the detailed numerical analysis and studies of the mixed-precision CholQR). Here, in Sect. 3.1, we first describe our implementation of the mixed-precision CholQR on the multicore CPUs with the GPU. Then, in Sect. 3.2, we present its performance. Finally, in Sect. 3.3, we study the effects of using the higher-precision on the performance of CA-GMRES.

3.1 Implementation

When the target hardware does not support a desired higher precision, software emulation is needed. For instance, double-double (dd) precision emulates the

double-double operation	# of double precision instructions Add/Substitute	Multiply	FMA	Total
Multiply (double-double input)	5	3	1	9
Multiply (double input)	3	1	1	5
Addition (IEEE-style)	20	0	0	20
Addition (Cray-style)	11	0	0	11

Fig. 4. Number of double-precision instructions in double-double operations.

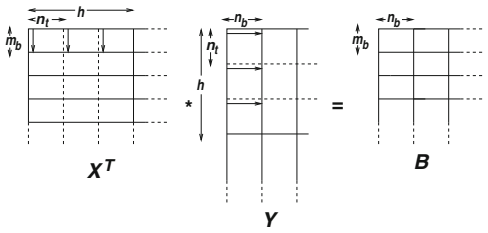


Fig. 5. *InnerProds* implementation (arrow shows data access by a GPU thread).

```
double regC[m_b][n_b], regA[m_b], regB
for ℓ = 1, ... h/n_t do
    for j = 1 ... n_b
      regA[i] = x_{ℓ*n_t,j}
    end for
    for j = 1, ..., n_b do
      regB = y_{ℓ*n_t,j}
      for i = 1 ... m_b
        regC[i][j] += regA[i] * regB
      end for
    end for
end for
```

Fig. 6. *InnerProds* pseudocode.

quadruple precision by representing each numerical value by an unevaluated sum of two double precision numbers, and is capable of representing the 106 bits precision, while the double-precision number is of 53 bits precision. There are two standard implementations [4] of adding two numerical values in double-double precision, $a + b = \widehat{c} + e$, where e is the round-off error; one satisfies the IEEE-style error bound ($e = \delta(a + b)$ with $|\delta| \leq 2\epsilon_{dd}$ and $\epsilon_{dd} = 2^{-105}$), and the other satisfies the weaker Cray-style error bound ($e = \delta_1 a + \delta_2 b$ with $|\delta_1|, |\delta_2| \leq \epsilon_{dd}$). Figure 4 lists the computational costs of the double-double operations required by our mixed-precision CholQR (dd-CholQR). The standard CholQR in double precision (d-CholQR) performs about a half of its total flops at Step 1 and the other half at Step 3. On the other hand, compared with the input matrix $V_{1:s+1}$, the Gram matrix B is much smaller in its dimension (i.e., $s \ll n$), and CholQR spends only a small portion of its flops and orthogonalization time, computing its Cholesky factor at Step 2. Hence, using the Cray-style double-double precision for Steps 1 and 2, our dd-CholQR performs about 8.5× more computation than d-CholQR. On the other hand, the increase in communication is less significant; our intra-GPU communication is about the same, only writing the s-by-s output matrix in double-double precision while reading and writing the n-by-s input matrix $V_{1:s+1}$ in double precision ($s \ll n$). In addition, we communicate twice more data between the GPUs ($16s^2$Bytes with $s \approx 10$), but with the same latency.

Though CholQR performs only a half of the total flops at Step 1, its orthogonalization time can be dominated by Step 1. This is because though

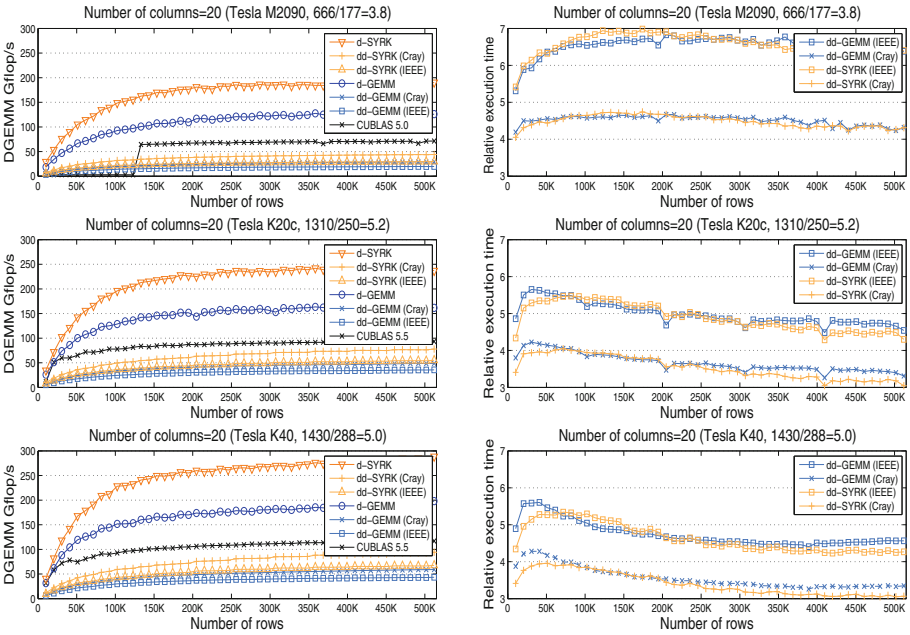


Fig. 7. Performance of standard and mixed-precision *InnerProds* in double precision.

the other half of the total flops is performed at Step 3, solving the triangular system with many right-hand-sides at Step 3 exhibits a high parallelism and can be implemented efficiently on a GPU. On the other hand, at Step 1, computing each element of the Gram matrix requires a reduction operation on two n-length vectors. These inner-products (*InnerProds*) are communication-intensive and exhibit only limited parallelism. Hence, Step 1 often becomes the bottleneck, where standard implementations fail to obtain high-performance on the GPU. In our *batched* implementation of a matrix-matrix multiply (GEMM) to compute *InnerProds*, $B := X^T Y$, each thread block computes a partial product, $B^{(i,j,k)} := X^{(k,i)T} Y^{(k,j)}$, where $X^{(k,i)}$ and $Y^{(k,j)}$ are the h-by-m_b and h-by-n_b blocks of X and Y, respectively.[1] Within the thread block, each of its n_t threads computes its partial result in the local registers (see Fig. 5 for an illustration, and Fig. 6 for the pseudocode, where $\underline{x}_{\ell,j}$ is the (ℓ, j)-th element of $X^{(k,i)}$). Then, each thread block performs the binary reduction of the partial results among its threads, summing n_r columns at a time using the shared memory to store $n_t \times (m_b \times n_r)$ numerical values. The final result is computed by another binary reduction among the thread blocks. Our implementation is designed to reduce the number of synchronizations among the threads while relying on the CUDA runtime and the parameter tuning to exploit the data locality. For the symmetric (SYRK) multiply, $B := V^T V$, the thread blocks compute only a triangular part

[1] In the current implementation, the numbers of rows and columns in X and Y are a multiple of h, and multiples of m_b and n_b, respectively, where n_b is a multiple of n_r.

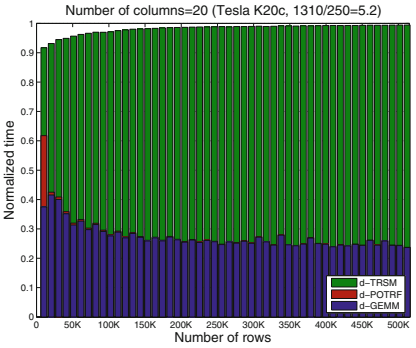


Fig. 8. d-CholQR time breakdown.

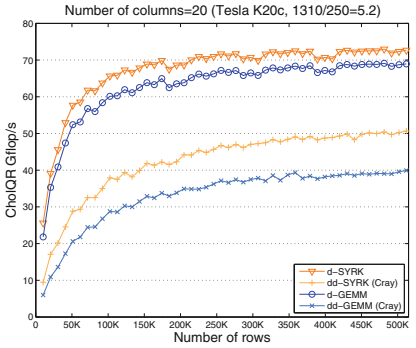


Fig. 9. d/dd-CholQR performance.

of B and reads $V^{(k,j)}$ once for computing a diagonal block. Our performance studies in the next subsection are based on this batched kernel.

3.2 Performance

Figure 7 compares the standard and mixed-precision *InnerProds* performance on different GPUs, where the mixed-precision *InnerProds* reads the input matrix in the standard 64-bits double precision, but accumulates its intermediate results into the output matrix in the double-double precision. Each GPU has a different relative cost of communication to computation, and on top of each plot, we show the ratio of the double-precision peak performance (Gflop/s) over the shared memory bandwidth (GB/s) (i.e., flop/B to obtain the peak). This ratio tends to increase on a newer architecture, indicating a greater relative communication cost. We tuned our kernel for each matrix dimension on each GPU in each precision (see the five tunable parameters h, m_b, n_b, n_r, and n_t in Sect. 3.1), and the figure shows the optimal performance. Based on the memory bandwidth and the fixed number of columns in the figure, the respective peak performances of the standard d-GEMM are 442, 625, and 720Gflop/s on the M2090, K20c, and K40 GPUs. Our d-GEMM obtained 29, 26, 28 % of these peak performances and speedups of about 1.8, 1.7, and 1.7 over CUBLAS 5.5 on these three GPUs. In addition, though it performs 16× more floating-point instructions, the gap between the standard d-GEMM and the mixed-precision dd-GEMM tends to decrease on a newer architecture, and dd-GEMM is only less than four times slower on K20c. We also see that by taking advantage of the symmetry, both d-SYRK and dd-SYRK improve the performance of d-GEMM and dd-GEMM, respectively.

Figure 8 shows the breakdown of the standard d-CholQR orthogonalization time on two eight-core Intel Sandy Bridge CPUs with one NVIDIA K20c GPU. Because of our efficient implementation of *InnerProds*, only about 30 % of the orthogonalization time is now spent in *d-InnerProds*. As a result, while the mixed precision *dd-InnerProds* was about four times slower than *d-InnerProds*, Fig. 9 shows that the mixed-precision dd-CholQR is only about 1.7 or 1.4 times slower

Name	Source	$n/1000$	nnz/n
cant	FEM Cantilever	62.4	64.2
shipsec1	FEM Ship secion	140.8	87.3
dielFilterV2real	FEM in Electromagnetic	1157.5	41.9
G3_circuit	Circuit simulation	1585.4	4.8

Fig. 10. Test matrices used for test cases with CA-GMRES.

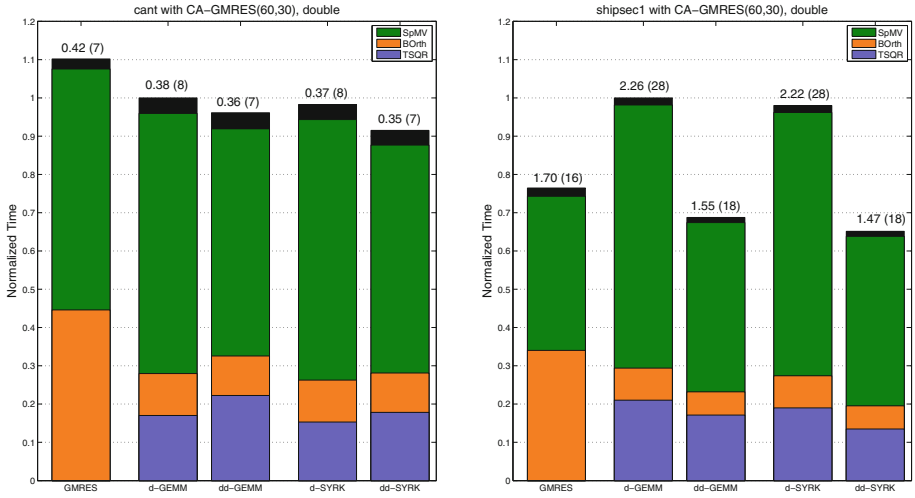


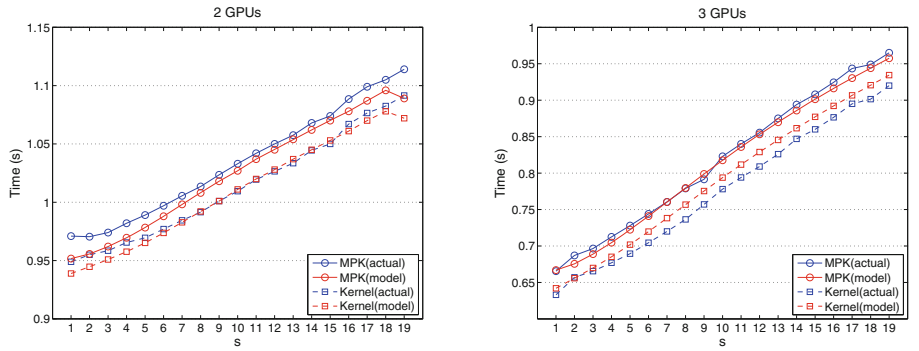
Fig. 11. Performance comparison of CA-GMRES using dd-CholQR (with dd-GEMM or dd-SYRK) against CA-GMRES using d-CholQR with (d-GEMM or d-SYRK) and GMRES using CGS: On top of each bar shows total time in seconds and restart count. To obtain the solution convergence, the reorthogonalization was used with d-CholQR, while it was not needed with dd-CholQR.

than the standard d-CholQR when GEMM or SYRK is used for *InnerProds*, respectively. In other words, if the reorthogonalization is avoided using the higher-precision, then dd-CholQR may obtain a performance competitive to that of d-CholQR with reorthogonalization. For the mixed-precision dd-CholQR, the Cholesky factorization in the double-double precision is computed using MPACK[2] on the CPU, while for d-CholQR, we use the threaded version of MKL for the Cholesky factorization in the double precision.

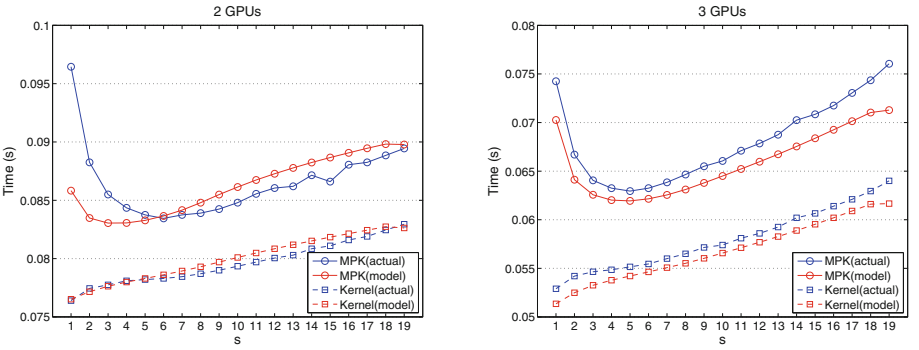
3.3 Case Studies with CA-GMRES

Figure 11 shows the solution time of CA-GMRES using the standard d-CholQR and the mixed-precision dd-CholQR on two eight-core Sandy Bridge CPUs with a single K20c. To maintain the numerical stability and obtain the solution convergence, the full-reorthogonalization was needed with d-CholQR, while it was

[2] http://mplapack.sourceforge.net.



(a) `dielFilterV2real` matrix



(b) `G3_Circuit` matrix

Fig. 12. Result of *MPK* performance model on two and three GPUs.

not needed with dd-CholQR. For *BOrth*, we used the classical Gram-Schmidt (CGS) process [3] with reorthogonalization, which obtains high performance with the GPU [8]. The solution time is normalized using the corresponding solution time of GMRES that uses CGS with reorthogonalization for orthogonalizing its basis vectors. Figure 10 shows the properties of our test matrices that were downloaded from the University of Florida Sparse Matrix collection.[3] We see that using dd-CholQR, even with the computationally expensive software emulation, the solution time was reduced not only because the reorthogonalization was avoided but also because CA-GMRES converged in fewer iterations.

4 Adaptive Step Size for Matrix Powers Kernel

Most of CA-GMRES implementations including ours [8] use the same step size s for *MPK*, *BOrth*, and *TSQR*, while the optimal s for *MPK* is typically smaller than that of *BOrth* or *TSQR* due to the computational and/or communication

[3] http://www.cise.ufl.edu/research/sparse/matrices/.

overheads associated with *MPK*. To address this performance difference, we first adapted our implementation such that *MPK* uses a smaller step size $\widehat{s}$ than the step size s used for *BOrth* and *TSQR*. Hence, to generate the s basis vectors, we invoke *MPK* $s/\widehat{s}$ times using the step size $\widehat{s}$ before calling *BOrth* and *TSQR*. In addition, instead of having a different $\widehat{s}$ for *MPK* as a user-specified input parameter, we design an adaptive scheme to dynamically adjust the step size $\widehat{s}$ of *MPK* based on the static inputs (i.e., the sparsity pattern of the coefficient matrix A and the maximum step size s) and the performance measurements gathered during the first restart-loop of CA-GMRES. In particular, for our experiments, we use the following performance model:

$$MPK \text{ time} = \text{Inter-communication time} + \text{Kernel time},$$

where we let

$$\text{Inter-communication time} = \text{Latency} + \frac{\text{Communication volume}}{\text{Bandwidth}}, \text{and}$$

$$\text{Kernel time} = \frac{\text{Flop count}}{\text{flop/s}} + \#\text{ of random data accesses} \times \text{ Data access time},$$

and "Kernel time" consists of the computation and intra-GPU communication time. In our experiments, "Communication volume" and "Flop count" are computed based on the sparsity pattern of the coefficient matrix A, while "# of random data accesses" is approximated by the aggregated number of non-local vector elements accessed by *MPK*. On the other hand, we computed "Latency," "Bandwidth," "flop/s," and "Data access time" based on the measured time of the reduction for the dot-products, point-to-point communication for *SpMV*, flop count and time required by *SpMV*, and data copy on the GPU, respectively. All the performance measurements are collected during the first restart loop of CA-GMRES. In practice, we often use GMRES iteration for the first restart loop (i.e., $s = 1$). This is because to maintain the numerical stability, *MPK* generates the Newton basis [1] whose shifts can be computed during the first restart. Since these shifts are not available for the first restart loop, to maintain the numerical stability, GMRES iteration is used. Hence, with the proposed adaptive scheme, we gather both the numerical and performance statistics of the given problem during the first restart loop. Then, based on these statistics, the input parameters are adjusted to enhance both the performance and stability of CA-GMRES for the remaining loops.

Figure 12 shows the effectiveness of the performance model to capture the performance of *MPK* for two sparse matrices on Intel Sandy Bridge CPUs with three NVIDIA Tesla M2090 GPUs. The properties of our test matrices from the University of Florida Sparse Matrix collection are shown in Fig. 10. Since we use the performance model to select a good step size, the model only needs to capture the performance trend and not the exact performance. In addition, in many cases, the performance of *MPK* does not change significantly around the optimal step size. The figure demonstrates that for both matrices, the model was successful in capturing the performance trends and selecting a near-optimal step

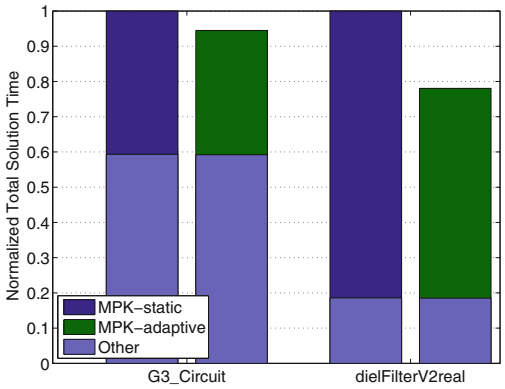


Fig. 13. Effects of adaptive *MPK* step size on performance of CA-GMRES.

size. In particular, for the `dielFilterV2real` matrix, due to the overhead associated with *MPK*, the standard *SpMV* was faster than *MPK*. Our performance model could capture this and select the step size of one for *MPK*.

Figure 13 shows the effects of the adaptive step size on the performance of CA-GMRES, where the static scheme uses the fixed step size for *MPK*, *BOrth*, and *TSQR* that obtains the near-optimal performance of CA-GMRES. Though the improvement was not significant, this is based on the near-optimal performance of *MPK*. We expect the benefit of the adaptive scheme to increase on the computer where the communication cost is higher (e.g., a GPU cluster). Finally, in all the test cases, it only required marginal overheads to gather the performance measurements.

5 Conclusion

We proposed a mixed-precision orthogonalization scheme to improve the numerical stability of CA-GMRES. When the target hardware does not support a desired higher precision, software emulation is needed. We showed that though the software emulation could significantly increase the computational cost, the increase in the communication cost is less significant. As a result, the overhead of using the software emulation is decreasing on a newer GPU architecture where the cost of the computation is decreasing compared to the cost of the communication. Our case studies on multicore CPUs with a GPU demonstrated that though it requires about 8.5× more computation, using a higher-precision for this small but critical segment of CA-GMRES can improve not only its overall stability but also, in some cases, its performance.

In this paper, we only studied the effects of a higher-precision on a single GPU. On multiple GPUs of a compute node, the performance of CA-GMRES depends more on the performance of the GPU kernels (i.e., intra-GPU communication) than the inter-GPU communication [8]. Hence, similar benefits of using a higher-precision are expected on the multiple GPUs. We will study its effects on a system with a greater communication latency (e.g., distributed GPUs or CPUs)

where the performance improvement by using the higher-precision arithmetic may be greater. We are also studying the use of mixed-precision in eigensolvers where the orthogonality can be more crucial, and are writing an extended paper focusing on the numerical properties of our mixed-precision scheme [9]. Finally, it is of our interest to apply or extend recent mixed precision efforts (e.g., reproducible BLAS[4] and precision tuning[5]) for our studies.

In this paper, we also studied an adaptive scheme to adjust the step size of *MPK* on multiple GPUs. Our performance results demonstrated that our adaptive scheme can find a near optimal step size based on the static input parameters and the performance measurements gathered during the first restart loop, and reduce the total solution time of CA-GMRES. Our *MPK* is currently optimized only for the inter-GPU communication which is relatively inexpensive on a node. We are looking to optimize *MPK* on a GPU, which should increase the benefit of the adaptive step size. We also plan to study the effectivness of the adaptive schme on a larger system with greater communication cost (e.g., a distributed system), where a greater benefit of the adaptive scheme is expected (in term of time or memory).

Acknowledgments. This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research (ASCR), and was founded in part by National Science Foundation under Grant No. ACI-1339822, DOE Grant #DE-SC0010042: "Extreme-scale Algorithms & Solver Resilience (EASIR)," Russian Scientific Fund, Agreement N14-11-00190, and NVIDIA. This research used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under Contract OCI-0910735. We thank Maho Nakata, Daichi Mukunoki, and the members of the DOE EASIR project for helpful discussions.

References

1. Bai, Z., Hu, D., Reichel, L.: A Newton basis GMRES implementation. IMA J. Numer. Anal. **14**, 563–581 (1994)
2. Demmel, J., Hoemmen, M., Mohiyuddin, M., Yelick, K.: Avoiding communication in computing Krylov subspaces. Technical report UCB/EECS-2007-123, University of California Berkeley EECS Department, October 2007
3. Golub, G., van Loan, C.: Matrix Computations, 4th edn. The Johns Hopkins University Press, Baltimore (2012)
4. Hida, Y., Li, X., Bailey, D.: Quad-double arithmetic: algorithms, implementation, and application. Technical report LBNL-46996 (2000)
5. Hoemmen, M.: Communication-avoiding Krylov subspace methods. Ph.D. thesis, University of California, Berkeley (2010)
6. Saad, Y., Schultz, M.: GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM J. Sci. Stat. Comput. **7**, 856–869 (1986)

[4] http://www.eecs.berkeley.edu/~hdnguyen/rblas.
[5] http://crd.lbl.gov/groups-depts/ftg/projects/current-projects/corvette/precimonious.

7. Stathopoulos, A., Wu, K.: A block orthogonalization procedure with constant synchronization requirements. SIAM J. Sci. Comput. **23**, 2165–2182 (2002)
8. Yamazaki, I., Anzt, H., Tomov, S., Hoemmen, M., Dongarra, J.: Improving the performance of CA-GMRES on multicores with multiple GPUs. In: The Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 382–391 (2014)
9. Yamazaki, I., Tomov, S., Dongarra, J.: Mixed-precision Cholesky QR factorization and its case studies on multicore CPUs with multiple GPUs. Submitted to SIAM J. Sci. Comput. (2014)

http://www.springer.com/978-3-319-17352-8

High Performance Computing for Computational Science -- VECPAR 2014
11th International Conference, Eugene, OR, USA, June 30 -- July 3, 2014, Revised Selected Papers
Daydé, M.; Marques, O.; Nakajima, K. (Eds.)
2015, XVII, 311 p. 146 illus., Softcover
ISBN: 978-3-319-17352-8